The File System Interface is an Anachronism

The File System Interface is an Anachronism

Daniel Ellard

TR-15-03

Computer Science Group
Harvard University
Cambridge, Massachusetts

The File System Interface is an Anachronism

Daniel Ellard
ellard@eecs.harvard.edu

November 18, 2003

Abstract

Contemporary file systems implement a set of abstractions and semantics that are suboptimal for many (if not most) purposes. The philosophy of using the simple mechanisms of the file system as the basis for a vast array of higher-level mechanisms leads to inefficient and incorrect implementations. We propose several extensions to the canonical file system model, including explicit support for lock files, indexed files, and resource forks, and the benefit of session semantics for write updates. We also discuss the desirability of application-level file system transactions and file system support for versioning.

1 Introduction

The canonical UNIX file system interface is elegant, ubiquitous, and suboptimal. The seductive appeal of such a clean and minimal abstraction is undeniable– and its simplicity and generality has unquestionably contributed to the success of operating systems such as UNIX [14], which in turn made this model pervasive.

Although we have gotten a lot of mileage out of the UNIX file system abstraction, there are many signs that this abstraction is no longer appropriate for contemporary application environments, and that something more general and flexible is required. Contemporary application domains require capabilities that are not provided by the file systems such as automatic archival, versioning, high availability, transaction support, global naming, replication, or fault tolerance. Some file system variants extend the basic file system to provide some of this functionality, but the most portable and widespread approach is for applications to provide the functionality they require via their own application-specific mechanisms. Unfortunately these are often designed with little thought of generality, and constructed in an ad-hoc or haphazard manner that leads to incomplete or buggy implementations.

This is a position paper, not a design document. We focus on defining the functionality that we believe will be necessary for future file systems, but we do not propose how this functionality should be implemented, nor what its interface should be. In some cases, there has already been much research and the only remaining question how to integrate the research results into commodity systems. In other cases, however, we're still haven't figured out exactly what we want, much less how to achieve it.

2 File System Abuse

The file system is routinely used in ways that are different from the original intent, and this leads to usage patterns for which the file system design is suboptimal, or difficult to get right. These abuses are not always

due to poor design decisions, but instead are often an inevitable consequence of the mismatch between what application developers need to do and the limits of the file system interface.

In this section we discuss three common abuses of the file system, and describe how they could be avoided in the future.

2.1 Interprocess Communication

The ability to modify the contents of a file in-place should not be used for synchronous interprocess communication. We propose that the standard UNIX write semantics be abandoned and replaced with session semantics. The advantage of this change to the UNIX file system semantics in the context of distributed file systems were touted in AFS [10], and has since been widely adopted. Concurrent applications that need to share data should use different mechanisms, such as those provided by *Linda* [4], *JavaSpace* [18], or the more recent (but unfortunately named) *Sparse Files* [19].

2.2 Lock Files

The use of files as a mutex or semaphore to protect resources is a misuse of the file system. A new type of file system object should be created to serve as a lock. The lock object has many of the same attributes as an ordinary file, such as an owner, modification and access times, and permission modes, but does not contain any data. The file system is free to optimize its storage and caching policies accordingly – it is even free to implement the lock in a completely different manner than files or directories, although for the sake of convenience we feel that it should share the same namespace as the ordinary file system.

2.3 Flat-File Tables

Flat-file tables or other objects with a naturally segmented internal structure should be stored in a manner that exposes this structure and allows both the file system and application to exploit it.

There are two common alternatives to the flat-file tables: using a file-per-record structure (where the table is implemented as a directory of files, each containing a single record) or using a database engine to store and access each record as a separate row in a database table. Each approach has merits, but also disadvantages: The file-per-record system can create directories that are hard to navigate or manipulate, and can be another source of inefficiency. Using a full-blown database, on the other hand, is overkill for many applications. There should be a middle ground – a simple indexed file type, using a mechanism akin to Berkeley DB [12] to provide access to records by key.

One frequent application of tables implemented as flat files is mailboxes. A recent study by Elprin & Parno shows that IMAP servers that use maildirs (implementing mailboxes as directories, where each message is stored in a separate file) or databases to store email outperform flat-file based IMAP servers by an order of magnitude for many operations [7].

An indexed file must be differentiated from an ordinary file for reasons of performance and ease of use. For the best performance, it is advantageous for the underlying file system to use different pre-fetching and caching strategies for indexed objects. This is particularly important for distributed file systems, such as NFS [11, 3], that do per-file caching. Any small change (such as appending a message to a flat-file mailboxe) to a file updates the modification time on the entire file and invalidates any cached copies, even though almost

all of the file has the same. For the case of active mailboxes this can lead to situations where the file cache is almost entirely ineffective [6].

For ease of use, it would be convenient for the object to appear as a flat-file if opened for reading using the conventional `open` interface, so that the wealth of text-based tools like `grep` can still be applied to this data. At the same time, however, it must be protected from ordinary writes – modifications to an indexed object must be made through an index-based API, rather than the ordinary `write` interface. Implementing this would not be very difficult, nor is it a huge departure from the current philosophy, at least not in the UNIX world – it is already possible to use `read` to view the names of items in a directory, but it is necessary to use other operations to get detailed information about files or modify the attributes of each item.

2.4 Discussion

One of the attractions of the original file system abstraction was its simplicity. Although these additions do add complexity to the interface and its implementation, the set of primitive objects is still small and the semantics of their operations are easy to specify and understand.

3 Higher-Level Functionality

3.1 Resource Forks

In file systems such as MacOS's HFS, each file has an associated *resource fork* that can be used to store arbitrary bindings between the file and data that can be used by applications or the OS. For example, in MacOS the preferred application to open or edit a specific file can be permanently associated with the file, and this information persists even if the file is copied, renamed, or relocated.

UNIX has a minimalist version of this; files can begin with magic numbers (*e.g.*, `#!`) that tell `exec` how to execute a program, or can be used by applications to guide how the file is interpreted. In Windows, the situation is even more dire.

It is not hard to imagine how to add a mechanism similar to resource forks to a UNIX-like file system. Every file and directory could have a resource file associated with it– we could split the inode space so that the inode number for each "real" file is even, and the inode number for the each corresponding resource file is simply the inode number for the "real" file plus 1. The resource file could contain arbitrary information about the file encoded in a standard format such as XML, such as the MIME-type of the file, its version number, and modification history.

3.2 Application-Level Transaction Support

Typical file systems do not have the ability to encapsulate a series of operations as a transaction. This functionality would be particularly useful for two reasons: first, to ensure that an observer always sees a consistent view of the file system, and second, to simplify application development.

Consistency is useful for applications that create or modify several files or directories but do not want any of these changes to be observed until all are finished. For example, during the installation of a new application and its configuration files, the system might pass through a state where the visible copy of the application and its configuration files are mutually incompatible, and an unlucky user who tries to run the

application at that moment might have their data clobbered. It would be advantageous to have updates that touch multiple files hidden inside a transaction and made visible only when complete.

Development of well-behaved and error-tolerant applications would be greatly simplified by the ability to "undo" a set of file system operations as a unit. In the best case, it is a tedious and error-prone task to keep track of all of the changes that the application makes to the file system and determine how to undo them if an error occurs. In the worst case, if a program is killed by an external signal or unanticipated error, even this work is in vain, and the file system can be left in an inconsistent state – and this inconsistency will persist until it is explicitly cleaned up. This problem can be completely avoided by encapsulating related changes to the file system inside transactions, so that only consistent states are visible.

3.3 Better Support for Versioning

The idea that support for versioning should be a central part of the file system is not new. In fact, it predates the current file system paradigms, but has been almost entirely replaced by application-level versioning systems such as CVS that run on top of the file system.

There have been many efforts to recreate this functionality, illustrating the belief that file system support for versioning leads to a more flexible, complete, and efficient system for preserving and restoring file system state. Unfortunately, these projects also demonstrate how little consensus there is about what the interface should look like:

- ClearCase [16] gives the user complete control over when versions are created, which objects are versioned, and which versions of objects are visible at any given moment.
- CVFS [17] records all changes to the file system within a fixed window of time, and discards the version history after the window has elapsed. At present, this is only used to provide complete analysis and replay or undo of recent changes, although they believe that their mechanism can be used in a more general system.
- Venti [13] records every change to the file system as a separate version, and keeps every version forever. It is not intended to be used as an interactive file system, however, but as the front end to a versioning and archival system.
- Elephant [15] attempts to automatically perform versioning and uses heuristics to decide which versions should be preserved and which can be discarded.
- WAFL [9] uses a purely time-based heuristic that preserves snapshots of the file system taken at fixed times for specific lengths of time. This is not versioning *per se*, but demonstrates the viability of the mechanism – given a user interface to control when snapshots are made, and what subset of the file system is contained in the snapshot, it could be used to build a powerful versioning system akin to ClearCase.

4 File Systems in Distributed Environments

he problems faced by distributed storage systems are more complex than those faced by monolithic systems [21]. As a result, the early protocols for distributed data services focused on utility, correctness and ease-of-implementation rather than generality. Unfortunately, these protocols are now firmly entrenched, and we

are saddled with an ever-growing number of highly-specialized and overlapping protocols. A typical UNIX client workstation in a networked configuration uses the following protocols to access remote information:

- DNS for hostname resolution
- NIS for workgroup information
- NNTP for access to news articles
- IMAP or POP3 for access to email messages
- SMTP to send email messages
- HTTP, FTP, or SCP to copy files to/from remote hosts
- NFS for workgroup file sharing

For a Windows workstation, some of the names are different, but the diversity and number of protocols is similar.

New protocols attempting to replace or augment some of these protocols, such as the *Lightweight Directory Access Protocol* (LDAP), or extensions to these protocols, such as WebDAV [22], appear regularly – and many of them fall into obscurity just as quickly, following in the footsteps of `archie`, `gopher`, `HyperFTP`, WAIS, the earlier POP protocols, and a myriad others. There are also frequent attempts to piggy-back one protocol on top of another, for example providing a subset of the NFS protocol via HTTP [2, 20] or even FTP [1, 5, 8].

The fact that people are going through such contortions to achieve seemingly straightforward functionality suggests that there's something missing from our current interfaces. More importantly, however, it is hard to imagine that we really need so many protocols when their capabilities and characteristics overlap in so many areas.

5 Conclusion

The UNIX local file system interface has survived almost without change for more than twenty years. It has become pervasive and outlived its predecessors. It is, by any measure, a great success, and will influence the design of countless future interfaces. However, our needs have evolved, and it is time for the file system interface to evolve as well. In this paper, we have identified several extensions to the interface that we believe deserve serious consideration: direct support for lock files, deprecation (or prohibition) of use of the file system as an IPC mechanism, support for simple indexed access methods for record-structured files, and adding "resource forks" to every file and directory These additions to the file system interface will streamline the development of new applications, as well as making them more efficient and robust.

We also discussed the issue of file system support for application-level transactions, and versioning, although we admit that the best way to approach each of these problems is much less obvious.

References

[1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the Operating System at the User-Level: the Ufo Global File System. In *Proceedings of the USENIX 1997 Technical Conference*, pages 77–90, 1997.

[2] Brent Callaghan. WebNFS: The File System for the Internet, April 1997. Sun Microsystems, Inc.

[3] Brent Callaghan, Brian Pawlowski, and Peter Staubach. NFS Version 3 Protocol Specification. `http://www.ietf.org/rfc/rfc1813.txt`, June 1995.

[4] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, September 1989.

[5] Vincent Cate. Alex – a Global File System. In *Proceedings of the USENIX File System Workshop*, pages 1–11, Ann Arbor, Michigan, 1992.

[6] Daniel Ellard and Margo Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Seventeenth Annual Large Installation System Administration Conference (LISA'03)*, pages 73–85, San Diego, CA, October 2003.

[7] Nicholas Elprin and Bryan Parno. An Analysis of Database-Driven Mail Servers. Technical Report TR-02-13, Harvard University DEAS, 2002.

[8] Deepak Gupta and Vikrant Sharma. Design and Implementation of a Portable and Extensible FTP to NFS Gateway. In *Proceedings of Principles and Practice of Programming in Java (PPPJ'02)*, Dublin, June 2002.

[9] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[10] J. Howard, M. Kazar, S. Menees, S. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[11] Bill Nowicki. NFS: Network File System Protocol Specification. `http://www.ietf.org/rfc/rfc1094.txt`, March 1989.

[12] Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the USENIX 1999 Technical Conference*, Monterey, CA, June 1999.

[13] S. Quinlan and S. Dorward. Venti: a New Approach to Archival Storage. In *First USENIX Conference on File and Storage Technologies*, pages 89–102, Monterey, CA, 2002.

[14] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[15] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Symposium on Operating Systems Principles*, pages 110–123, 1999.

[16] Atria Software. ClearCase User's Manual, 1994.

[17] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. Technical Report CMU-CS-02-145, Carnegie Mellon University, School of Computer Science, May 2002.

[18] Sun Microsystems, Inc. JavaSpace Specification, 1998. http://java.sun.com/products/jini/specs.

[19] Douglas Thain and Miron Livny. The Case for Sparse Files. Technical Report 1464, University of Wisconsin, Computer Sciences Department, January 2003.

[20] Amin Vahdat, Tom Anderson, Mike Dahlin, Eshwar Belani, David Culler, Paul Eastham, and Chad Yoshikawa. WebOS: Operating System Services for Wide Area Applications. In *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, 1998.

[21] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997.

[22] E. James Whitehead, Jr. and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the Web. In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)*, pages 291–310, Copenhagen, Denmark, September 1999.